Analyzing Dynamic Call Graphs Enhanced with Program State Information for Feature Location and Understanding

Johannes Bohnet, Jürgen Döllner
University of Potsdam
Hasso-Plattner-Institute
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
{bohnet, doellner}@hpi.uni-potsdam.de

ABSTRACT

In this paper we present a prototype tool for locating and understanding features in unfamiliar source code of complex C/C++ software systems. The key concepts of the analysis tool are (1) combining dynamic function call graphs with information on the functions' containment within hierarchically structured modules and (2) providing means of gathering program state information on user-defined locations within the call graph. Starting from a typically huge dynamic call graph logged during feature execution, developers narrow their search for functions implementing the feature's core functionality by successively pruning the graph from feature-irrelevant functions. The assessment whether a function contributes to a feature is performed on various levels of abstraction, namely on (1) module containment, (2) on the function's call relations, and (3) on the way it modifies data. Having sorted out feature-irrelevant functions this way, users are able to analyze the remaining, probably highly feature-relevant functions with other heavy-weight analysis techniques, such as debugging techniques, to get a deep understanding of the feature implementation.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering

General Terms

Design, Human Factors.

Keywords

Maintenance, Reverse Engineering, Feature Location, Concept Assignment, Software Visualization, Dynamic Analysis.

1. MOTIVATION

All successful software gets changed as successful software frequently needs to be adapted to altered user requirements and changed system environments. Requests for changes are typically expressed in terms of features, i.e., in system functionalities that are triggered by users and that produce user-visible outputs. To fulfill a feature change request, developers need to trace the feature down to design artifacts, which is typically a very time- and therefore cost-intensive task. In this paper we present a prototype tool for supporting developers in locating features within large amount of unfamiliar C/C++ source code. Additionally to just identifying feature relevant code artifacts, the tool supports developers in getting a first understanding of how the artifacts interact to produce feature functionality.

ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

2. ANALYSIS PROCESS

Figure 1 illustrates the key ideas of the analysis process with our prototype tool. The following steps need to be performed by a developer using the tool:

1) Identification of an execution scenario that executes the system's feature the developer needs to modify or extend. If possible, the execution scenario should consist of a sequence of user-system interactions that triggers the feature in isolation.

2) Execution of the feature while function call graph logging is enabled. The analysis tool thereupon combines the function call graph with the hierarchy of modules automatically.

3) Exploration of the call graph with the techniques provided by the tool. Goal is to semi-automatically prune the call graph from parts that do not contribute to the feature. The pruning process is mainly supported by (a) showing the graph's function call relations within a higher-level context, i.e., the module hierarchy, and (b) by providing code representations that are enhanced with code coverage data and dynamically resolved function calls.

4) Frequently, the developer reaches points, where from control flow analysis it is difficult to decide whether an executed function represents feature relevant code or not. The tool therefore provides a mechanism to selectively gather program state information by setting breakpoints for automatically reading out specific data in the next run.

5) Feature re-execution. This time additionally to call graph logging, the requested information on the program state is collected when the system's control flow passes a breakpoint.

6) Iteration of steps 3 to 5 until the function call graph is reduced to those control paths that most likely represent the execution of the feature's core functionality.

7) The now moderately sized call graph and its participating functions are target of heavy-weight analysis techniques, such as debugging techniques, to get a deep understanding of the feature implementation.

3. CALL GRAPH EXPLORATION

Our prototype tool provides an elaborate interactive visualization front-end that supports developers in exploring call graphs. The key idea is to guide the user along feature-relevant

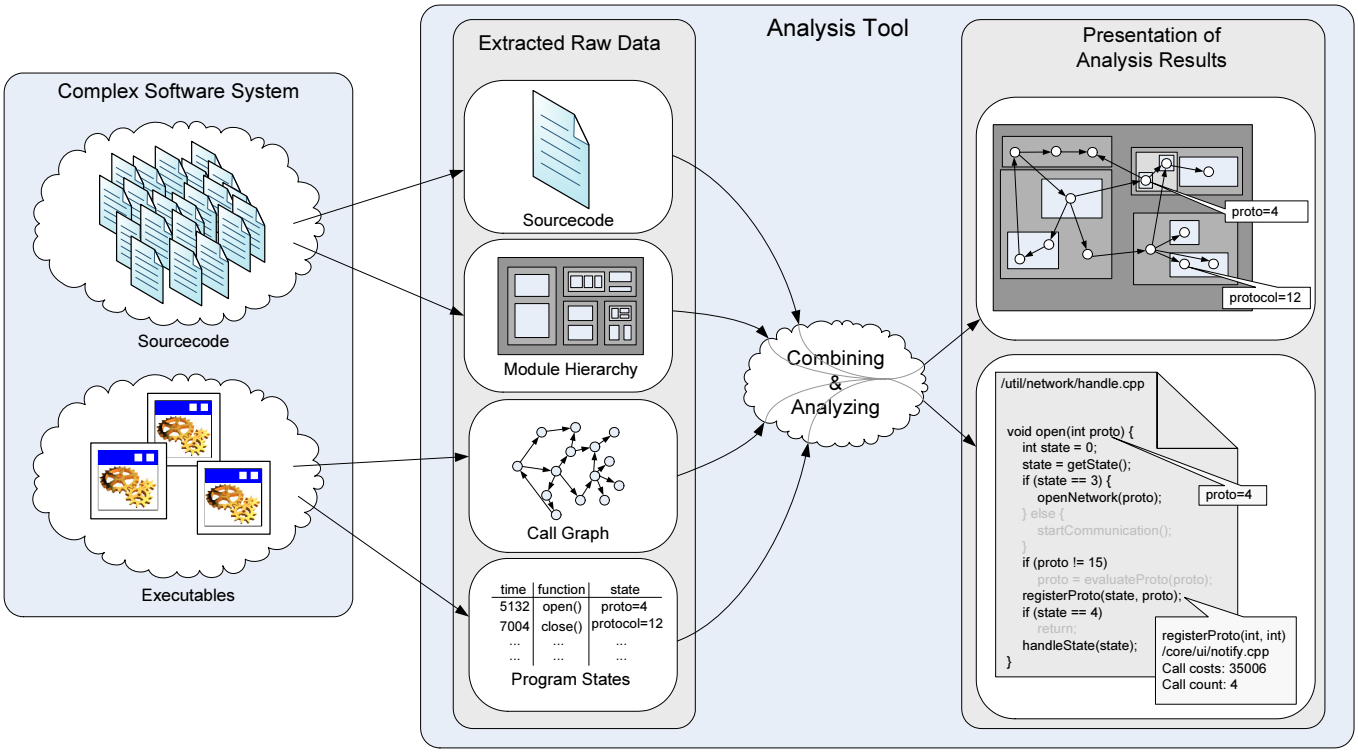

Figure 1: The analysis tool extracts both static and dynamic facts from the software system under analysis. Combined and filtered facts are thereupon visualized in multiple views for being efficiently explored.

control-paths by preparing subgraphs of the complete call graph that best give contextual information on a function currently in the developer's focus of interest. Contextual information is provided as different type of data on various levels of abstraction:

1) The module hierarchy where a function is located in;
2) Control-paths entering or leaving the function;
3) The function's code representation additionally enhanced with dynamically resolved function calls and with code coverage information. This way, code is visually pruned from unexecuted parts that are not necessary for understanding feature execution. Actually, they are likely to be hampering understanding the currently analyzed run;
4) States of selected variables/parameters/objects.

Selected concepts of the visualization front-end are described in detail in some of our earlier publications [1, 2, 3, 4]. These concepts include elaborate interaction techniques with the graph visualization system and smart level-of-detail techniques for annotating graphical representations with text labels. The latter – smart labeling techniques – is a crucial factor for user acceptance of graph visualization systems because such techniques bridge the gap between graphical representations and the data they represent.

4. LIMITS OF THE APPROACH

The tool supports developers during their first analysis steps when having to perform a feature change request on a complex software system with millions of lines of unfamiliar code. It helps to rapidly identify those parts of the code that most likely implement the feature. It also gives a first picture of how these code parts interact to produce feature functionality. For a deep understanding, however, heavy-weight analysis techniques such as debugging techniques are necessary. Although our tool copes with complex production code, an obstacle for using the tool in the developer's daily routine is the tool's status as stand-alone application. For being fully useful and accepted by developers, it is important for a program comprehension tool that it is smoothly integrated within existing developing and maintenance processes. That's why we plan to integrate the tool as plug-in in various integrated development environments.

5. REFERENCES

[1] Bohnet, J. and Döllner, J. 2006. Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems. In Proceedings of the ACM Symposium on Software Visualization, pp. 95-104.

[2] Bohnet, J. and Döllner, J. 2007. Visually Exploring Control Flow Graphs to Support Legacy Software Migration. In Proceedings of the GI Conference on Software Engineering SE07, pp. 245-246.

[3] Bohnet, J. and Döllner, J. 2007. Facilitating Exploration of Unfamiliar Source Code by Providing 2-1/2-D Visualizations of Dynamic Call Graphs. In Proceedings of the IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis VISSOFT, pp. 63-66.

[4] Bohnet, J. and Döllner, J. 2007. CGA Call Graph Analyzer - Locating and Understanding Functionality within the Gnu Compiler Collection's Million Lines of Code. In Proceedings of the IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis VISSOFT, pp. 161-162.